

Deciding Monotone Duality and Identifying Frequent Itemsets in Quadratic Logspace

Georg Gottlob

Department of Computer Science
University of Oxford, Oxford OX1 3QD, UK
georg.gottlob@cs.ox.ac.uk

Abstract

The monotone duality problem is defined as follows: Given two monotone formulas f and g in nonredundant DNF, decide whether f and g are dual. This problem is the same as duality testing for hypergraphs, that is, checking whether a hypergraph \mathcal{H} consists of precisely all minimal transversals of a simple hypergraph \mathcal{G} . By exploiting a recent problem-decomposition method by Boros and Makino (ICALP 2009), we show that duality testing for hypergraphs, and thus for monotone DNFs, is feasible in $\text{DSPACE}[\log^2 n]$, i.e., in quadratic logspace. As the monotone duality problem is equivalent to a number of problems in the areas of databases, data mining, and knowledge discovery, the results presented here yield new complexity results for those problems, too. For example, it follows from our results that whenever for a Boolean-valued relation (whose attributes represent items), a number of maximal frequent item sets and a number of minimal infrequent item sets are known, then it can be decided in quadratic logspace whether there exist additional frequent or infrequent set.

Keywords: Duality testing, frequent item set, hypergraph, transversal, data mining.

1 Introduction

This paper derives new complexity bounds for the problem DUAL of deciding whether two irredundant monotone Boolean formulas in DNF are mutually dual, or, equivalently, of deciding whether two simple hypergraphs are dual, i.e., whether each of these hypergraphs consists precisely of the minimal transversals of the other. While the exact complexity remains open, there is progress: We prove a $\text{DSPACE}[\log^2 n]$ upper bound for DUAL, and another, presumably tighter bound that is expressed in terms of sophisticated machine-bounded complexity classes. The DUAL problem is actually one of the most mysterious problems in theoretical computer science. It has many applications, especially in the database, data mining, and knowledge discovery areas [5, 6, 19], some of which will be mentioned below. Let us first describe the DUAL problem more formally.

Duality testing for monotone DNFs and hypergraphs. Two Boolean formulas $f(x_1, x_2, \dots, x_n)$ and $g(x_1, x_2, \dots, x_n)$ on propositional variables x_1, x_2, \dots, x_n are *dual* if

$$f(x_1, x_2, \dots, x_n) \equiv \neg g(\neg x_1, \neg x_2, \dots, \neg x_n).$$

A monotone DNF is *irredundant* if the set of variables in none of its disjuncts is covered by the variable set of any other disjunct. The *duality testing problem* DUAL is the problem of testing whether two irredundant monotone DNFs f and g are dual.

A *hypergraph* \mathcal{H} is a finite family of finite sets (also called *hyperedges*) defined over some set of *vertices* $V(\mathcal{H})$. By default, if $V(\mathcal{H})$ is not explicitly specified, the set of vertices of \mathcal{H} is $\bigcup_{E \in \mathcal{H}} E$. A *transversal* of \mathcal{H} is a subset of $V(\mathcal{H})$ that meets all hyperedges of \mathcal{H} , and a *minimal transversal* of

\mathcal{H} is a transversal of \mathcal{H} that does not contain any other transversal as subset. The set of all minimal transversals of a hypergraph \mathcal{H} is denoted by $tr(\mathcal{H})$. The *Hypergraph Duality Problem* is the problem of deciding for two simple hypergraphs \mathcal{G} and \mathcal{H} whether $\mathcal{G} = tr(\mathcal{H})$. In case $\mathcal{G} \neq tr(\mathcal{H})$, to witness this, one may want to exhibit a *new transversal of \mathcal{H} with respect to \mathcal{G}* . This is a transversal of \mathcal{H} that has no hyperedge of \mathcal{G} as subset. Obviously, every new transversal H contains at least one *new minimal transversal of \mathcal{H} w.r.t. \mathcal{G}* , but it needs not to be minimal itself.

It is well-known that DNF duality and hypergraph duality are actually the same problem (see [5]). In fact, two irredundant monotone DNFs f and g are dual iff their hypergraphs are dual. The hypergraph associated to a monotone DNF has precisely one hyperedge for each disjunct, consisting of the set of all variables of this disjunct. Vice versa, one can trivially associate an irredundant DNF to each hypergraph and thus reduce hypergraph duality to DNF duality. Given that these problems essentially coincide (and can be reduced to each other via trivial reductions that are much easier than logspace reductions), we regard them as one and the same problem, which we refer to as DUAL.

The duality problem in data mining, database theory, and knowledge discovery. Most prominently, the DUAL problem is at the core of a number of important data mining and database problems. It is central, for example, to the determination of the maximal frequent and minimal infrequent sets in data mining. More precisely, consider a Boolean-valued data relation M over a set S of attributes called *items*, and a threshold z with $0 < z \leq |M|$. Each subset $U \subseteq S$ is called an *itemset*. For each tuple t of M , let $items(t) = \{A \in S \mid t[A] = 1\}$. The frequency $f(U)$ for an itemset U is the number of tuples t of M , such that $U \subseteq items(t)$. U is *frequent* if $f(U) > z$ and *infrequent* otherwise. In data mining, one is interested in computing the *maximal frequent sets* and the *minimal infrequent sets* (under set inclusion) for M and z . Let us refer to the former as $IS^+(M, z)$ and to the latter as $IS^-(M, z)$. Clearly, both $IS^+(M, z)$ and $IS^-(M, z)$ are simple hypergraphs over S , and we abbreviate them by IS^+ and IS^- , respectively, when M and z are understood. As a fundamental result towards the aim of computing IS^+ and IS^- , it was shown in [19] that the minimal frequent itemsets are exactly the minimal transversals of the complements of the maximal infrequent itemsets, i.e. $IS^- = tr(\overline{IS^+})$, and thus also $IS^+ = tr(\overline{IS^-})$, where for $A \subseteq 2^S$, $\overline{A} = \{S - A \mid A \in S\}$.

Let MAXFREQ-MININFREQ-IDENTIFICATION be the following decision problem in data mining: Given M , z , a set $\mathcal{G} \subseteq IS^-(M, z)$, and a set $\mathcal{H} \subseteq IS^+(M, z)$, decide whether $\mathcal{H} = IS^+(M, z)$ and $\mathcal{G} = IS^-(M, z)$, that is, whether there exists no additional maximal frequent or minimal infrequent itemsets for M and z , that is not already in $\mathcal{G} \cup \mathcal{H}$. In [19] it was shown that there exist no such additional itemsets iff $\mathcal{G} = tr(\overline{\mathcal{H}})$. With regard to the computational complexity, we thus have:

Proposition 1.1 ([19]). MAXFREQ-MININFREQ-IDENTIFICATION is logspace-equivalent to DUAL.

The results of [19], are at the base of a host of algorithms for maximal frequent itemset generation, that compute both IS^+ and IS^- incrementally. These algorithms initialize \mathcal{G} and $\overline{\mathcal{H}}$ with some easy to compute subsets of IS^- and $\overline{IS^+}$, respectively. Then, at each step they check whether for the current sets $\mathcal{G} = tr(\overline{\mathcal{H}})$ is true, and if not, compute one or more new transversals from which new maximal frequent itemsets or minimal infrequent itemsets can be computed easily (see e.g. [1, 27]). Thus, not only the decision problem DUAL is of relevance to data mining, but also the problem of effectively computing a new transversal that acts a witness that $\mathcal{G} \neq tr(\overline{\mathcal{H}})$. In the present paper, we will obtain results on the complexity of this latter problem, too.

Another interesting related database problem is the ADDITIONAL KEY FOR INSTANCE problem for explicitly given relational instances. Given a relational instance R over attribute set S , and a set K of minimal keys for R , determine if there exists a key for R that is not already contained in K . This problem, which has been shown equivalent to DUAL in the early nineties [5], may be of renewed interest in our times of Big Data, where we are faced with massive data tables.

Proposition 1.2 ([5]). The ADDITIONAL KEY FOR INSTANCE problem is logspace equivalent to

$\overline{\text{DUAL}}$. Moreover, enumerating the minimal keys of a relational instance R is equivalent to enumerating the set $\text{tr}(\mathcal{H})$ for some hypergraph \mathcal{H} which is logspace-computable from R .

Other related problems equivalent to DUAL or to $\overline{\text{DUAL}}$ deal with the construction of Armstrong relations for sets of functional dependencies [5], see also [17, 4].

Let us finally briefly mention a completely different problem from the area of distributed databases. For quorum-based updates [24] in distributed databases, the concept of *coterie*, which is essentially a hypergraph of intersecting quorums has been introduced, and one is specifically interested in so called *non-dominated coterie*s (for definitions and details, see [12, 20]). The following was proven:

Proposition 1.3 ([20, 5]). *A coterie \mathcal{H} is non-dominated iff $\text{tr}(\mathcal{H}) = \mathcal{H}$.*

There are a large number of applications of the DUAL problem and of hypergraph dualization in the areas of knowledge discovery, machine learning, and more generally in AI and knowledge representation. Just to mention a few: Learning monotone Boolean CNFs and DNFs with membership queries [19], model-based diagnosis [26, 18], computing a Horn approximation to a non-Horn theory [22, 15], and computing minimal abductive explanations to observations [8]. Surveys of these and other applications and further references can be found in [6, 5].

Known complexity results. The exact complexity of DUAL has remained an open problem. Fredman and Khachiyan [11] have shown that DUAL is in $\text{DTIME}[n^{o(\log n)}]$, more precisely, that it is contained in $\text{DTIME}[n^{4\chi(n)+O(1)}]$, where $\chi(n)$ is defined by $\chi(n)^{\chi(n)} = n$. Eiter, Gottlob, and Makino [7], and independently, Kavvadias and Stavropoulos [23] have shown that DUAL is in the complexity class $\text{co-}\beta_2\text{P}$, which means that showing that the complement of DUAL can be solved in polynomial time with $O(\log^2 n)$ nondeterministic bits. This small amount of nondeterminism can actually be improved to $O(\chi(n) \log n)$ which is $o(\log^2 n)$, see [7].

Research question tackled The question about the space-efficiency of DUAL, namely, whether DUAL can be solved space-efficiently has not been satisfactorily answered since it was posed several times since 1995, for example in [5, 28, 9]. We believe that this is an important question which may actually turn out to be of practical relevance when mining large data sets with terabytes of data. This is the main problem we tackle. In addition, we aim at obtaining a better understanding of the DUAL problem in terms of machine-based structural complexity.

Results. We show in this paper that DUAL is indeed in $\text{DSpace}[\log^2 n]$, which is a very low class in POLYLOGSPACE . From this, modulo the widely believed assumption that $\text{PTIME} \not\subseteq \text{POLYLOGSPACE}$ we thus obtain satisfactory evidence that DUAL is not PTIME -hard, which answers another complexity question posed in [5, 9]. Our results are based on a careful analysis of a recent problem decomposition Method by Boros and Makino [2]. Their decomposition method actually yields a parallel algorithm that solves DUAL on an EREW PRAM in $O(\log^2)$ time using $n^{\log n}$ processors. However, it is currently not known whether such EREW PRAMS can be simulated in $\text{DSpace}[\log^2 n]$, and this is actually considered to be rather unlikely. However, Boros' and Makino's algorithm does not seem to exploit the full potential of a PRAM, and by taking into account the restricted pattern of information flow imposed by the specific self-reductions used in their algorithm, we succeeded to show membership of DUAL in $\text{DSpace}[\log^2 n]$.

Complexity theorists have very good reasons to assume that the space class $\text{DSpace}[\log^2 n]$ is incomparable with respect to containment to the class $\text{co-}\beta_2\text{P}$. It is thus somewhat unsatisfactory to have two upper bounds for DUAL that are incomparable, which suggests that, most likely, there exist better bounds. This encouraged us to look for a tighter upper bound for DUAL in terms of machine-based complexity models, that would be contained in both $\text{DSpace}[\log^2 n]$ and $\text{co-}\beta_2\text{P}$, and we succeeded to find one. We can, in fact, show that $\overline{\text{DUAL}}$ belongs to the "guess and check" class $\text{GC}(\log^2 n, \llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log})$. This somewhat exotic new machine-based complexity class contains precisely all problems that can be solved by first guessing $O(\log^2 n)$ bits and then checking

the correctness of this guess in $\llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log}$, which is a complexity class contained in PTIME we will define in the present paper. We hope that this tighter new bound will provide a better insight into the very nature of the DUAL problem, and possibly hint at the right direction for future research towards finding a matching upper bound.

Roadmap. The paper is organised as follows. In the next section we discuss decomposition methods for DUAL and give a succinct description of the method of Boros and Makino, which we consider to be the currently most advanced method. In Section 3, we define complexity classes based on iterated self-compositions of functions and prove a useful complexity-theoretic lemma. In Section 4, we use this lemma to prove our main result, namely that DUAL is in $\text{DSpace}[\log^2 n]$. Finally, in section 5 we provide our tighter structural complexity bound for DUAL. The paper is concluded in Section 6, where we also exhibit a diagram (see Fig. 1 on page 11) that puts all relevant complexity classes in relation, and highlights the new upper bounds.

2 The Decomposition Method by Boros and Makino

Most algorithms for deciding DUAL rely on decompositions that start with an original DUAL instance and recursively transform it into a conjunction of smaller instances, until each instance is either seen to be a no-instance because it violates necessary conditions for duality, or until it is small and efficiently decidable. Such decompositions are also known as *self-reductions*, see, e.g., Section 5.3 of [7]. The decomposition process corresponds in the obvious way to a *decomposition tree*. Different decomposition methods give rise to decomposition trees of different shapes and depths. For example, the well-known algorithm A by Fredman and Khachiyan [11] produces a "skinny" binary decomposition tree of depth linear in the input volume $|\mathcal{G}| \times |\mathcal{H}|$, while their algorithm B produces a non-binary tree of similar depth, but with fewer nodes. Later, decomposition methods giving rise to trees of polylogarithmic depth were published. In particular, the methods of Kavvadias and Stavropoulos [23] as well as the two methods by Elbassioni in [10] give rise to decomposition trees of polylogarithmic depth. Finally, decomposition methods yielding trees of logarithmic depth were presented by Gaur [13] (see also Gaur and Krishnamurti [14]), and, more recently, by Boros and Makino [2]. As we will show, the logarithmic-depth decomposition trees generated by these methods can be used to show that DUAL is in $\text{DSpace}[\log^2 n]$. In particular, we use the elegant decomposition method of Boros and Makino [2] to prove this, but we could have used Gaur's [13] in a similar fashion. In the rest of this section, we give a succinct description of the method of Boros and Makino, that contains all the essentials we need for our subsequent complexity analysis. It is assumed that the input instance $I = (\mathcal{G}, \mathcal{H})$ we have $|\mathcal{H}| \leq |\mathcal{G}|$, and that $\mathcal{G} \subseteq \text{tr}(\mathcal{H})$ and $\mathcal{H} \subseteq \text{tr}(\mathcal{G})$. Clearly this can be tested in logarithmic space.

For an input instance $I = (\mathcal{G}, \mathcal{H})$ of DUAL over a vertex set V , let $T(\mathcal{G}, \mathcal{H})$ denote its decomposition tree. Let $\mathbb{N}_{\mathcal{H}} = \mathbb{N}^0 \cup \mathbb{N}^1 \cup \mathbb{N}^2 \cup \dots \cup \mathbb{N}^{\lfloor \log |\mathcal{H}| \rfloor}$, where \mathbb{N} are the natural numbers and \mathbb{N}^0 stands for the sequence \emptyset . Thus $\mathbb{N}_{\mathcal{H}}$ contains all sequences of natural numbers of length up to $\lfloor \log |\mathcal{H}| \rfloor + 1$.

Each node of $T(\mathcal{G}, \mathcal{H})$ has five data structures associated with it:

- (i) A unique label $\text{label}(\alpha)$ consisting of a sequence $\alpha \in \mathbb{N}_{\mathcal{H}}$. In particular, the root α_0 of $T(\mathcal{G}, \mathcal{H})$ is labeled by \emptyset , and the i -th child of a node labeled (j_1, \dots, j_k) is labeled (j_1, \dots, j_k, i) .
- (ii) A set $S_\alpha \subseteq V(\mathcal{G})$.
- (iii) An instance of DUAL $\text{inst}(\alpha) = (\mathcal{G}^{S_\alpha}, \mathcal{H}_{S_\alpha})$,
where $\mathcal{G}^{S_\alpha} = \{E \cap S_\alpha \mid E \in \mathcal{G}\}$ and $\mathcal{H}_{S_\alpha} = \{E \in \mathcal{H} \mid E \subseteq S_\alpha\}$.
- (iv) A marking $\text{mark}(\alpha) \in \{\text{DONE}, \text{FAIL}, \text{NIL}\}$, where each leaf of the decomposition tree will be marked with DONE or FAIL, and each non-leaf is marked with dummy value NIL. Intuitively, each leaf marked DONE identifies a branch that does not contradict $\mathcal{H} = \text{tr}(\mathcal{G})$, whereas a leaf marked FAIL identifies a branch that proves that $\mathcal{H} \neq \text{tr}(\mathcal{G})$.

- (v) A set of vertices $t(\alpha) \subseteq V(\mathcal{G})$. This set will be the empty set for each node not marked FAIL, and, in case α is marked FAIL, will contain a witness for $\mathcal{H} \neq tr(\mathcal{G})$ in form of a new transversal of \mathcal{G} with respect to \mathcal{H} .

Let us now describe the method for building $T(\mathcal{G}, \mathcal{H})$ and deciding whether $\mathcal{H} = tr(\mathcal{G})$ in detail. At each stage of the algorithm, let us denote the set of current leave nodes by Λ . Here is how the tree is built. The input instance $(\mathcal{G}, \mathcal{H})$ is first transformed into a initial tree consists of the root α_0 with $label(\alpha_0) = \emptyset$, $S_{\alpha_0} = V$, $inst(\alpha_0) = (\mathcal{G}, \mathcal{H})$, $mark(\alpha_0) = \text{NIL}$, and $t(\alpha_0) = \emptyset$. At each stage of the decomposition, first, each leaf $\alpha \in \Lambda$ where $|H_{S_\alpha}| \leq 1$, will be marked by the following procedure, and will then not be further expanded and will thus be a leaf of the final tree $T(\mathcal{G}, \mathcal{H})$:

PROCEDURE MARKSMALL(α):

- CASE 1. IF $|H_{S_\alpha}| = 0$ and $\emptyset \notin \mathcal{G}^{S_\alpha}$, THEN $\{ mark(\alpha) := \text{FAIL}; t(\alpha) := S_\alpha \}$.
- CASE 2. IF $|H_{S_\alpha}| = 0$ and $\emptyset \in \mathcal{G}^{S_\alpha}$, THEN $\{ mark(\alpha) := \text{DONE}; t(\alpha) = \emptyset \}$.
- CASE 3. IF $H_{S_\alpha} = \{H\}$ and $\{\{i\} | i \in H\} \subseteq \mathcal{G}^{S_\alpha}$, THEN $\{ mark(\alpha) := \text{DONE}; t(\alpha) = \emptyset \}$.
- CASE 4. OTHERWISE let $mark(\alpha) := \text{FAIL}$, and let $t(\alpha) := S_\alpha - \{i\}$ for some arbitrarily chosen $i \in H$ with $\{i\} \notin \mathcal{G}^{S_\alpha}$.
-

Then, each leaf α of Λ not yet marked is subjected to the following procedure:

PROCEDURE PROCESS(α):

1. Let I_α consist of those vertices of \mathcal{H}_{S_α} that occur in more than $|\mathcal{H}_{S_\alpha}|/2$ hyperedges of \mathcal{H}_{S_α} ;
 2. IF I_α is a new transversal of \mathcal{G}^{S_α} with respect to \mathcal{H}_{S_α} , THEN
 $\{ mark(\alpha) := \text{FAIL}; t(\alpha) := (V - S_\alpha) \cup I_\alpha$; EXIT PROCEDURE};
 3. OTHERWISE IF there is a $G \in \mathcal{G}^{S_\alpha}$ such that $G \cap I_\alpha = \emptyset$ THEN let

$$\mathcal{C} = \{S_\alpha - (E - \{i\}) | E \in \mathcal{G}_G^{S_\alpha} \text{ and } i \in E \cap G\},$$
where $\mathcal{G}_G^{S_\alpha} = \mathcal{G}^{S_\alpha} - \{E' \in \mathcal{G}^{S_\alpha} | E' \subseteq S_\alpha - G\}$;
 4. OTHERWISE IF there exists a $H \in \mathcal{H}_{S_\alpha}$ such that $H \subseteq I_\alpha$ THEN let

$$\mathcal{C} = \{S_\alpha - \{i\} | i \in H\} \cup \{H\}$$
;
 5. Let $\kappa(\alpha) = |\mathcal{C}|$ and the elements of \mathcal{C} be $C_1, C_2, \dots, C_{\kappa(\alpha)}$. For each C_i , $1 \leq i \leq \kappa(\alpha)$, create a new child α_i with $label(\alpha_i) = (label(\alpha), i)$,
 $S_{\alpha_i} = C_i$, $inst(\alpha_i) = (\mathcal{G}^{S_{\alpha_i}}, \mathcal{H}_{S_{\alpha_i}})$, $mark(\alpha_i) = \text{NIL}$, and $t(\alpha_i) = \emptyset$.
-

Exhaustively apply the procedures MARKSMALL (to unmarked leaves α having $|H_{S_\alpha}| \leq 1$) and PROCESS (to all other unmarked leaves), until there are no unmarked leaves left in the tree. The resulting tree is then $T(\mathcal{G}, \mathcal{H})$. The following proposition summarizes results by Boros and Makino [2].

Proposition 2.1 (Boros and Makino [2]).

1. $\mathcal{H} = tr(\mathcal{G})$ iff all leaves of $T(\mathcal{G}, \mathcal{H})$ are marked DONE.
2. The depth of $T(\mathcal{G}, \mathcal{H})$ is bounded by $\log |\mathcal{H}|$.
3. Each node α of $T(\mathcal{G}, \mathcal{H})$ has at most $|V| \cdot |\mathcal{G}|$ children, i.e., $\kappa(\alpha) \leq |V| \cdot |\mathcal{G}|$, where V is the set of vertices of \mathcal{G} and \mathcal{H} .
4. In case $\mathcal{H} \neq tr(\mathcal{G})$, $t(\alpha)$ is a new transversal of \mathcal{G} with respect to \mathcal{H} .

3 A Complexity-Theoretic Lemma

For numerical function¹ z , we denote by $\text{DSPACE}[z(n)]$ ($\text{FDSPACE}[z(n)]$) the class of all all decision problems (computation problems) solvable deterministically in $O(z(n))$ space. For a function f , let $f^1 = f$ and for $i \geq 1$, let $f^{i+1} = f \circ f^i$, where \circ is the usual function composition, i.e., where for each x in the domain of g , $(f \circ g)(x) = f(g(x))$. Let Q denote the set of all functions computable in space $O(\log^2 n)$ from strings over an input alphabet to the non-negative natural numbers, and let

Q_{\log} denote that subclass of Q containing all functions ρ , where for each input string I , $\rho(I)$ is $O(\log |I|)$. For each function $\rho \in Q$, let f^ρ denote the function that to each input I associate the output $f^\rho(I) = f^{\rho(I)}(I)$. If FC denotes a functional complexity class, then $[\text{FC}]^{\log}$ denotes the class of functions that can be built from some function f in FC via a logarithmic number $\rho(I) + O(\log n)$ of self-compositions of f for each input of size n :

$$[\text{FC}]^{\log} = \bigcup_{f \in \text{FC}, \rho \in Q_{\log}} \{f^\rho\}.$$

For a functional complexity class FC , the subclass FC_{pol} is given by all functions f of FC for which there exists a polynomial γ such that for each input I , and for each $1 \leq i$, $|f^i(I)| \leq \gamma(|I|)$. Note that for many classes FC , FC_{pol} is a proper subclass of FC . This is, for example, the case for $\text{FDSPACE}[\log n]$, i.e., functional logspace. For instance, let f be the function that associates to an input of size n an output consisting of n^2 zeroes. Clearly, $f \in \text{FDSPACE}[\log n]$, but the output sizes of the f^i are not bounded by any fixed polynomial when i varies.

Lemma 3.1. $[\text{FDSPACE}[\log n]_{\text{pol}}]^{\log} \subseteq \text{FDSPACE}[\log^2 n]$.

Proof. The proof is similar the well-known proof that for any two functions $f, g \in \text{FDSPACE}[\log n]$, their composition $g \circ f$ is in $\text{FDSPACE}[\log n]$, too, see, e.g. [25]). However, here, the logarithmic (rather than constant) number of compositions is responsible for the blowup of the required space by a logarithmic factor. Let f be a function from strings to strings in $\text{FDSPACE}[\log n]_{\text{pol}}$, realized by a logspace Turing Machine T , and let $\rho \in Q_{\log}$. In order to prove the lemma, it is sufficient to show that one can construct a single functional Turing machine T^* with space bound $O(\log^2 n)$, that for each input I of length n , simulates the pipelined application $T^{\rho(I)}$ that outputs $f^{\rho(I)}(I)$. T^* simulates an arrangement of $\rho(I)$ copies of T , say, $T_1, T_2, \dots, T_{\rho(I)}$, such that the input string v_1 to T_1 is I , and such that for $i \geq 1$, the input string v_{i+1} to T_{i+1} is equal to the output string w_i of T_i . Given that the size of $w_i = T^i(I)$ is bounded by some fixed polynomial γ , there are numbers a and b such that each T_i requires no more than space $a + b \log n$. When simulating the pipelined computation $T_{\rho(I)}(T_{\rho(I)-1}(\dots(T_2(T_1(I))))$ on a single Turing machine T^* , we have to avoid the effective storage of any intermediate output w_i (or, equivalently, input v_{i+1}). To this aim, T^* simulates each T_i via a logspace procedure P_i that maintains its own space area on the worktape of T^* . Each P_i acts like T_i , except for the following modifications: For $1 < i < \rho(I)$, P_i has a single output bit which is stored on the worktape of T^* ; moreover P_i takes as input a dedicated special index register d_i that specifies which output bit of T_i is to be computed, and computes only this output bit (suppressing all other output bits) and stores it in a single-bit register o_i . T_i 's access to its j -th input bit is then simulated by P_i writing " j " (in binary) into the special index register d_{i-1} , starting P_{i-1} , and then waiting until P_{i-1} writes the desired output bit into o_{i-1} which corresponds to the correct value of the j -th output of T_{i-1} , and thus the j -th input bit to T_i . P_1 and $P_{\rho(I)}$ work in a similar way, except that P_1 directly accesses the input string I from the input tape of T^* , and $P_{\rho(I)}$, rather than suppressing some output bits, writes *all* output bits to the output tape of T^* .

The workspace required by each procedure P_i is easily seen to be bounded by $a' + b' \log n$ for some fixed constants a' and b' independent of n . This reflects the $a + b \log n$ bits required to execute T_i , plus

¹We only consider time and space constructible functions here.

the little extra space P_i may require for its index d_i , for the output bit o_i , and for a constant number of auxiliary counters and pointers (of size at most $a + b \log n$ bits each) for control and stack management for the P_i procedures. Given that $\rho(I)$ is $O(\log n)$, T^* requires $O(\log^2 n)$ space in total. \square

Note that the same space bound doesn't hold for $\llbracket \text{FDSPACE}[\log n] \rrbracket^{\log}$. In fact, with functions f in this class, intermediate outputs $f^i(I)$ may be of superpolynomial size, and in the worst case, even of exponential size $n^{\Theta(n)}$. Therefore, when omitting the “pol” restriction, the best space bound we are able to show is $\llbracket \text{FDSPACE}[\log n] \rrbracket^{\log} \subseteq \text{PSPACE}$. Since an $\text{FDSPACE}[\log^2 n]$ Turing machine has an output of size at most $n^{O(\log n)}$, it is actually the case that $\llbracket \text{FDSPACE}[\log n] \rrbracket^{\log} \not\subseteq \text{FDSPACE}[\log^2 n]$.

4 The Space Bound for DUAL

The main result of this section is that for a pair $(\mathcal{G}, \mathcal{H})$, the entire decomposition tree $T(\mathcal{G}, \mathcal{H})$ (with all markings and labels) produced by the decomposition method of Boros and Makino as outlined in Section 2 can be computed with quadratic logspace. The other space-complexity results follow from this as simple corollaries.

We start with a lemma that provides us with a logarithmic space bound for computing the i -th child of a node α of the decomposition tree from the fully labeled node α and from the set V of vertices of the original input instance, or for discovering that such a child does not exist. If α is a node of the decomposition tree, let us denote by $\text{attr}(\alpha)$ the attributes of α , i.e., the tuple $(\text{label}(\alpha), S_\alpha, \text{inst}(\alpha), \text{mark}(\alpha), t(\alpha))$.

Lemma 4.1. *There is a deterministic logspace procedure $\text{NEXT}(V, \text{attr}(\alpha), i)$, which for each DUAL instance $(\mathcal{G}, \mathcal{H})$ over vertex set V , for each attribute set $\text{attr}(\alpha)$ of a node α of $T(\mathcal{G}, \mathcal{H})$, and for each positive integer $i \leq |V| \cdot |\mathcal{G}|$ outputs:*

- $\text{attr}(\alpha_i)$ if α_i is the i -th child of α in $T(\mathcal{G}, \mathcal{H})$;
- IMPOSSIBLE otherwise (i.e., if α has less than i children).

Proof. First note that by simple inspection it is immediate that the procedures MARKSMALL and PROCESS given in Section 2 can be implemented by deterministic logspace transducers. In fact, these procedures only perform simple cardinality checks, counting, assignments, and set theoretic operations that are all well-known to run in logspace. A procedure NEXT, as required, can be constructed as follows. If $\text{label}(\alpha) \in \{\text{DONE}, \text{FAIL}\}$ then output IMPOSSIBLE, otherwise perform the composition $\text{MARKSMALL}^*(\text{PROCESS}^*(\alpha))$, where:

- PROCESS^* works like PROCESS except that it outputs only the i -th child of α , if such a child exists, rather than outputting all children, and output IMPOSSIBLE otherwise; and
- MARKSMALL^* works like MARKSMALL, except that it also accepts the input IMPOSSIBLE, in which case it also outputs IMPOSSIBLE.

These minor modifications of MARKSMALL and PROCESS clearly run in deterministic logspace, therefore, also their composition does, and hence so does the procedure NEXT. \square

A *path descriptor* for a DUAL instance $I = (\mathcal{G}, \mathcal{H})$ over a vertex set V is a list of length $\leq \lceil \log |\mathcal{H}| \rceil$, whose elements are integers bounded by $|V| \cdot |\mathcal{G}|$. The set of all path descriptors for I is denoted by $PD(I)$. Clearly, $PD(I) \subset \mathbb{N}_{\mathcal{H}}$, and each label $\text{label}(\alpha)$ of a node α of $T(\mathcal{G}, \mathcal{H})$ is contained in $PD(I)$. Intuitively, a path descriptor, exactly in the same way as a label, is intended to describe a sequence of child-indices, that, starting from the roof of $T(\mathcal{G}, \mathcal{H})$ lead to a specific node α

of $T(\mathcal{G}, \mathcal{H})$. The root of $T(\mathcal{G}, \mathcal{H})$ is identified by the empty path descriptor. If $\pi = (i_1, i_2, \dots, i_r)$ is a path descriptor, then $\text{head}(\pi) = i_1$ and $\text{tail}(\pi)$ is the path descriptor (i_2, \dots, i_r) . Two path descriptors of the form (i_1, \dots, i_r) and $(i_1, \dots, i_r, i_{r+1})$ are said to be *consecutive*.

Lemma 4.2. *There is a procedure $\text{PATHNODE}(I, \pi)$ that runs in deterministic space $O(\log^2(|I|))$, that for each DUAL input instance I and path descriptor $\pi \in PD(I)$ outputs $\text{attr}(\alpha)$ if π corresponds to the label $\text{label}(\alpha)$ of a node in $T(\mathcal{G}, \mathcal{H})$, and outputs WRONGPATH otherwise.*

Proof. Let $I = (\mathcal{G}, \mathcal{H})$, $V = V(\mathcal{G})$, and $\pi \in PD(I)$, and let $\ell(\pi)$ denote the length of the sequence π (recall that $\ell(\pi) \leq \log |I|$). The procedure PATHNODE first computes in deterministic logspace $\text{Attr}(\alpha_0)$ for the root α_0 of $T(\mathcal{G}, \mathcal{H})$. It then computes $f^{\ell(\pi)}(V, \text{attr}(\alpha_0), \pi)$, where f is the function corresponding to the procedure F described as follows. F accepts as input either the string WRONGPATH , or a triple (W, attr, γ) where W is a set, attr is a data structure of the same format as the attributes $\text{attr}(\beta)$ of some vertex β in a decomposition tree, and π is a sequence of positive integers. On all other inputs, F outputs the empty string. On input WRONGPATH , F outputs WRONGPATH ; otherwise F computes $F'(\text{NEXT}(W, \text{attr}, \text{head}(\gamma)))$, where NEXT be as specified in Lemma 4.1, and where F' is as follows. F' outputs WRONGPATH if $\text{NEXT}(W, \text{attr}, \text{head}(\gamma)) = \text{IMPOSSIBLE}$, and F' outputs $(W, \text{Attr}', \text{tail}(\gamma))$, whenever $\text{NEXT}(W, \text{attr}, \text{head}(\gamma)) = \text{Attr}'$ for some attribute description Attr' . Since NEXT runs in deterministic logspace, also F' and F do, and therefore f is a logspace computable function.

By construction and by Lemma 4.1, PATHNODE precisely computes the attributes $\text{Attr}(\alpha)$ if there is a node α with $\text{label}(\alpha) = \pi$ in $T(\mathcal{G}, \mathcal{H})$, whereas otherwise PATHNODE outputs IMPOSSIBLE . Since $\ell(\pi)$ is clearly in Q_{\log} , by Lemma 3.1, $f^{\ell(\pi)}(V, \text{attr}(\alpha_0), \pi)$ can be computed in deterministic space $O(\log^2 n)$, and so can therefore PATHNODE . \square

By using a procedure PATHNODE according to the above Lemma, we are now ready to formulate an algorithm DECOMPOSE that computes the decomposition tree $T(\mathcal{G}, \mathcal{H})$ to a DUAL instance $(\mathcal{G}, \mathcal{H})$. In particular, the algorithms first lists the vertices and then the edges of the tree $T(\mathcal{G}, \mathcal{H})$.

Algorithm DECOMPOSE :

Input: DUAL-instance $I = (\mathcal{G}, \mathcal{H})$; Output: $T(\mathcal{G}, \mathcal{H})$.

BEGIN

OUTPUT("Vertices:");

FOR each path descriptor $\pi \in PD(I)$ DO

IF $\text{PATHNODE}(I, \pi) \neq \text{WRONGPATH}$ THEN OUTPUT($\text{PATHNODE}(I, \pi)$);

OUTPUT("Edges:");

FOR each pair π, π' of consecutive path descriptors π, π' in $PD(I)$ DO

BEGIN

$\alpha := \text{PATHNODE}(I, \pi)$;

$\alpha' := \text{PATHNODE}(I, \pi')$;

IF $\alpha' \neq \text{WRONGPATH}$ THEN OUTPUT($\langle \text{label}(\alpha), \text{label}(\alpha') \rangle$);

END

END.

Theorem 4.1. *The Algorithm DECOMPOSE computes the decomposition tree $T(\mathcal{G}, \mathcal{H})$ to a DUAL instance $(\mathcal{G}, \mathcal{H})$ in space $O(\log^2 n)$.*

Proof. The correctness of the algorithm follows from the correctness of PATHNODE as shown in Lemma 4.2. For the space bound, note that each path descriptor requires only $O(\log^2 |I|) = O(\log^2 n)$ bits, and that we can thus iterate (by re-using workspace) over all path descriptors and pairs of path descriptors in $O(\log^2 n)$ space. Given that, by Lemma 4.2, PATHNODE also runs in $O(\log^2 n)$ space, the entire DECOMPOSE algorithm needs only $O(\log^2 n)$ space. \square

Corollary 4.1.

1. *Deciding DUAL is in DSPACE[log² n].*
2. *If $tr(\mathcal{G}) \neq \mathcal{H}$, then computing a new transversal of \mathcal{G} with respect to \mathcal{H} is in FDSpace[log² n].*

Proof. In both cases we can first compute the entire decomposition tree $T(\mathcal{G}, \mathcal{H})$ in FDSpace[log² n], and then (i) for problem 1 check by a DLOGSPACE procedure whether all leaves are marked DONE, and (ii) for problem 2, use an FLOGSPACE procedure to find a node α labeled FAIL in $T(\mathcal{G}, \mathcal{H})$ and output its component $t(\alpha)$. Let \circ denote the composition operator for complexity classes in the obvious sense. Given that $\text{FDSpace}[\log^2 n] \circ \text{DLOGSPACE} = \text{DSpace}[\log^2 n]$, and given that, moreover, $\text{FDSpace}[\log^2 n] \circ \text{FLOGSPACE} = \text{FDSpace}[\log^2 n]$, the complexity bounds follow. Alternatively, we can solve the problems 1 and 2 directly by respective slight modifications of DECOMPOSE. \square

Note that if $tr(\mathcal{G}) \neq \mathcal{H}$, the witness $t(\alpha)$ produced is not necessarily a *minimal* transversal of \mathcal{G} , but is, in general, just transversal of \mathcal{G} that contains no edge of \mathcal{G} and thus witnesses that $tr(\mathcal{G}) \neq \mathcal{H}$, because $t(\alpha)$ must *contain* a missing minimal transversal of \mathcal{G} . From $tr(\alpha)$ such a minimal transversal t can easily be computed in polynomial time by letting first $t := t(\alpha)$ and by then successively eliminating vertices v from t for which $t - \{v\}$ is still a transversal of \mathcal{G} . However this process requires linear space in the vertex set V to remember the eliminated vertices plus logarithmic space in the instance size $|(\mathcal{G}, \mathcal{H})|$ for checking. This is still better than polynomial space in the full instance size, but is not quite in quadratic logspace. It is currently not clear whether there exists a smarter algorithm that requires quadratic logspace only.

5 Tightening the complexity bound

By the results of the previous section, DUAL and its complement $\overline{\text{DUAL}}$ are in quadratic logspace, i.e., in $\text{DSpace}[\log^2 n]$. On the other hand, as already mentioned, the complement of DUAL is in $\beta_2\text{P}$, the class of problems solvable in polynomial time with $O(\log^2 n)$ nondeterministic guesses. This class is identical with the class $\text{GC}(\log^2 n, \text{P})$ of the so called *Guess and Check* model of limited nondeterminism [3, 16], where $O(\log^2 n)$ nondeterministic bits are guessed before the proper computation starts and are appended to the input.

Given that the class P is generally believed to be incomparable with $\text{DSpace}[\log^2 n]$ (cf [21]), and given that $\text{P} \subseteq \text{GC}(\log^2 n, \text{P})$, it is very likely that also $\text{GC}(\log^2 n, \text{P})$ and $\text{DSpace}[\log^2 n]$ are incomparable. Since DUAL belongs to both classes, this suggests that neither well characterizes DUAL, and that DUAL is unlikely to be complete for either. This observation incited us to look out for a complexity class containing DUAL that would be contained in both $\text{GC}(\log^2(n), \text{P})$ and $\text{DSpace}[\log^2 n]$, and that would thus constitute a tighter upper complexity bound for DUAL than all those we have seen so far. In this section, we present precisely such a complexity class. In order to describe this class, we state some definitions and prove a lemma.

Let C be a complexity class and s a numerical function. Then $\text{GC}(s(n), C)$ is the class of all languages L for which there exists a language $A \in C$ such that an input string I is in L iff there is a string J of $O(s(|I|))$ bits, such that (I, J) is in A . In other words, L is in $\text{GC}(s(n), C)$ iff the membership of a string I in L can be checked in C after having guessed $O(s(n))$ nondeterministic bits that can be used as an additional input. The class $\llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log}$ is defined as the composition of $\llbracket \text{FDSpace}[\log n]_{\text{pol}} \rrbracket^{\log}$ with $\text{LOGSPACE} = \text{DSpace} \log n$, formally:

$$\llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log} = \llbracket \text{FDSpace}[\log n]_{\text{pol}} \rrbracket^{\log} \circ \text{LOGSPACE}.$$

Thus, an input I is first transformed to an output O by a $\llbracket \text{FSPACE} \log n_{\text{pol}} \rrbracket^{\log}$ procedure, after which O is submitted to a LOGSPACE decision procedure which will decide based on O if the original input I is accepted or rejected ².

Lemma 5.1. *Given a DUAL instance $I = (\mathcal{G}, \mathcal{H})$ and a path descriptor $\pi \in PD(I)$, deciding whether $\text{PATHNODE}(I, \pi)$ outputs a leaf of $T(\mathcal{G}, \mathcal{H})$ whose mark-component is FAIL is in $\llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log}$.*

Proof. The proof of Lemma 4.2 already shows that PATHNODE is in $\llbracket \text{FSPACE}[\log n] \rrbracket^{\log}$. Deciding whether $\text{PATHNODE}(I, \pi)$ outputs a leaf of $T(\mathcal{G}, \mathcal{H})$ whose mark-component is FAIL can thus be implemented by first executing $\text{PATHNODE}(I, \pi)$, and then checking whether the output is a node labeled FAIL. This is obviously in $\llbracket \text{FSPACE}[\log n] \rrbracket^{\log} \circ \text{LOGSPACE} = \llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log}$. \square

Finally, we study the main class of this section: $\text{GC}(\log^2 n, \llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log})$.

Theorem 5.1. $\overline{\text{DUAL}} \in \text{GC}(\log^2 n, \llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log})$.

Proof. In order to find a new transversal t of \mathcal{G} with respect to \mathcal{H} for a DUAL instance $I = (\mathcal{G}, \mathcal{H})$, rather than computing the entire decomposition tree $T(\mathcal{G}, \mathcal{H})$, it is sufficient to guess a branch of this tree that terminates in a leaf α labeled FAIL, and then compute $t(\alpha)$. Guessing such a branch amounts to guess a path descriptor π and then checking that $\text{PATHNODE}(I, \pi)$ outputs a node marked FAIL. Guessing π amounts to guess $\log^2 n$ bits, and this is all our guess-and-check algorithm guesses. Checking that π is a FAIL node is, by Lemma 5.1 in $\llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log}$, hence the overall computation can be done within $\text{GC}(\log^2 n, \llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log})$. \square

The last theorem of this section shows, as promised, that $\text{GC}(\log^2 n, \llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log})$ is effectively a subclass of the other tightest lower bounds that are most likely incomparable to each other: $\text{DSpace}[\log^2 n]$ and $\text{GC}(\log^2 n, P) = \beta_2 P$.

Theorem 5.2. $\text{GC}(\log^2 n, \llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log}) \subseteq \text{DSpace}[\log^2 n] \cap \text{GC}(\log^2 n, P)$.

Proof. For the inclusion $\text{GC}(\log^2 n, \llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log}) \subseteq \text{DSpace}[\log^2 n]$, note that a decision procedure in $\text{GC}(\log^2 n, \llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log})$ amounts to (i) guessing $O(\log^2 n)$ bits, which can be simulated by an exhaustive enumeration of all possible guesses (under re-use of space), which is feasible in $\text{DSpace}[\log^2 n]$, and (ii) for each such simulated guess, performing a check that lies in the complexity class $\llbracket \text{FSPACE}[\log n]_{\text{pol}} \rrbracket^{\log} \circ \text{LOGSPACE}$. Since, by Lemma 1, $\llbracket \text{FSPACE}[\log n]_{\text{pol}} \rrbracket^{\log} \subseteq \text{DSpace}[\log^2 n]$, and given that the composition of a function in $\text{DSpace}[\log^2 n]$ with a LOGSPACE computation yields a $\text{DSpace}[\log^2 n]$ computation, the overall computation is in $\text{DSpace}[\log^2 n]$.

To establish the inclusion $\text{GC}(\log^2 n, \llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log}) \subseteq \text{GC}(\log^2 n, P)$, it is sufficient to see that $\llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log} \subseteq \text{PTIME}$. This is the case. A decision procedure in $\llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log}$ amounts to a pipelined execution of $O(\log n)$ instantiations of a logspace function f , where the intermediate results are guaranteed to be of polynomial size in the original input, followed by the application of a logspace Boolean procedure g . This can be replaced by the pipelined execution of $O(\log n)$ instances of a PTIME procedure equivalent to f , followed by the application of a Boolean PTIME procedure equivalent to g . In total, this latter process is in PTIME because it amounts to a logarithmic number of invocations of a PTIME procedure, where each time the input size is bounded by a polynomial in the size n of the overall input. Therefore, $\llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log} \subseteq \text{PTIME}$. \square

²Note that $\llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log}$ is by all means a complexity class defined in terms of machines and resource bounds. In addition to the classical resources such as the amount of workspace, we here involve somewhat more unusual resources such as the allowed number of self-compositions, which is here bounded by $O(\log n)$, whence the superscript \log , and the allowed size of intermediate outputs in compositions, which is here polynomially bounded, whence the subscript pol .

6 Summary, discussion, and conclusion

In this paper we have derived new complexity bounds for the DUAL (or $\overline{\text{DUAL}}$) problem that show that these problems can, in principle, be implemented by space-efficient algorithms. These bounds are depicted in Figure 1 in relation to the other relevant complexity classes. Here, set-inclusion is visualized by ascending lines or paths. We believe that our results represent some progress in the long and rather tortuous battle towards a better understanding of the mysterious DUAL problem.

We do not claim that our results have immediate practical consequences, but we actually do hope that these bounds will prove useful. Firstly, the $O(\log^2 n)$ space bound indicates that there exist space-efficient algorithms, and this encourages us to look for *practical* space efficient solution methods for DUAL and its equivalent problems in data mining and in the database area. We feel that space-efficiency may be an advantage, when dealing with big data stemming from financial transactions or biological experiments and so on. When mining terabytes of data, we might want to trade workspace (i.e., main memory) for runtime. Our results state that this is, in principle, feasible. Future re-

search will show, whether our space bound can be exploited to come up with a reasonable algorithm. Secondly, our results may serve as a guide for research towards a matching bound for the DUAL problem. We have reasons not to believe that DUAL is hard for $\text{co-GC}(\log^2 n, \llbracket \text{LOGSPACE}_{\text{pol}} \rrbracket^{\log})$. This upper bound, however, gives us some intuition of where to dig further.

Acknowledgments: The author is grateful to E. Allender, E. Boros, K. Makino, E. Malizia, P. Rossmanith and H. Vollmer for help with technical questions and references.

References

- [1] E. Boros, V. Gurvich, L. Khachiyan, and K. Makino, *On the complexity of generating maximal frequent and minimal infrequent sets*, Proceedings 19th Symposium on Theoretical Aspects of Computing (STACS '99), LNCS, no. 2285, 2002, pp. 133–141.
- [2] Endre Boros and Kazuhisa Makino, *A fast and simple parallel algorithm for the monotone duality problem*, ICALP (1) (Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikolettseas, and Wolfgang Thomas, eds.), Lecture Notes in Computer Science, vol. 5555, Springer, 2009, pp. 183–194.
- [3] Liming Cai and Jianer Chen, *On the amount of nondeterminism and the power of verifying (extended abstract)*, MFCS (Andrzej M. Borzyszkowski and Stefan Sokolowski, eds.), Lecture Notes in Computer Science, vol. 711, Springer, 1993, pp. 311–320.
- [4] János Demetrotovics and Vu Duc Thi, *Armstrong relations, functional dependencies and strong dependencies*, Computers and Artificial Intelligence **14** (1995), no. 3.
- [5] Thomas Eiter and Georg Gottlob, *Identifying the minimal transversals of a hypergraph and related problems*, SIAM J. Comput. **24** (1995), no. 6, 1278–1304.
- [6] ———, *Hypergraph transversal computation and related problems in logic and ai*, JELIA (Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni, eds.), Lecture Notes in Computer Science, vol. 2424, Springer, 2002, pp. 549–564.

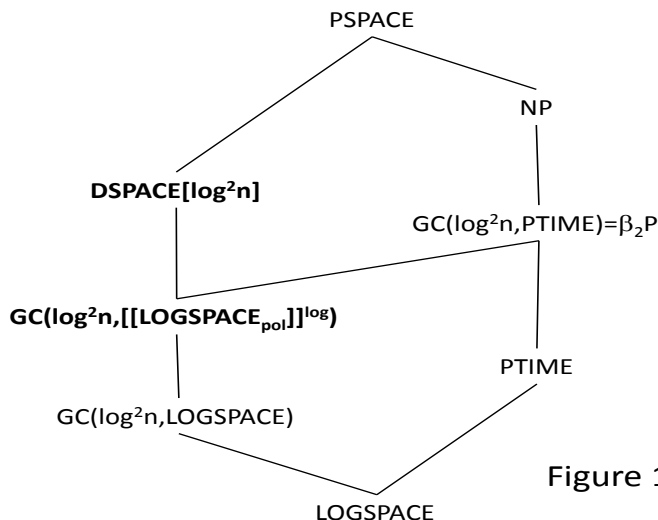


Figure 1

- [7] Thomas Eiter, Georg Gottlob, and Kazuhisa Makino, *New results on monotone dualization and generating hypergraph transversals*, SIAM J. Comput. **32** (2003), no. 2, 514–537.
- [8] Thomas Eiter and Kazuhisa Makino, *Generating all abductive explanations for queries on propositional horn theories*, CSL (Matthias Baaz and Johann A. Makowsky, eds.), Lecture Notes in Computer Science, vol. 2803, Springer, 2003, pp. 197–211.
- [9] Thomas Eiter, Kazuhisa Makino, and Georg Gottlob, *Computational aspects of monotone dualization: A brief survey*, Discrete Applied Mathematics **156** (2008), no. 11, 2035–2049.
- [10] Khaled M. Elbassioni, *On the complexity of monotone dualization and generating minimal hypergraph transversals*, Discrete Applied Mathematics **156** (2008), no. 11, 2109–2123.
- [11] M.L. Fredman and L. Khachiyan, *On the Complexity of Dualization of Monotone Disjunctive Normal Forms*, Journal of Algorithms **21** (1996), 618–628.
- [12] Hector Garcia-Molina and Daniel Barbará, *How to assign votes in a distributed system*, J. ACM **32** (1985), no. 4, 841–860.
- [13] D.R. Gaur, *Satisfiability and self-duality of monotone Boolean functions*, Dissertation, School of Computing Science, Simon Fraser University, January 1999.
- [14] D.R. Gaur and R. Krishnamurti, *Self-duality of bounded monotone boolean functions and related problems*, Proceedings 11th International Conference on Algorithmic Learning Theory (ALT), LNCS, no. 1968, 2000, pp. 209–223.
- [15] G. Gogic, Ch. Papadimitriou, and M. Sideri, *Incremental Recompile of Knowledge*, Journal of Artificial Intelligence Research **8** (1998), 23–37.
- [16] Judy Goldsmith, Matthew A. Levy, and Martin Mundhenk, *Limited nondeterminism*, SIGACT News **27** (1996), no. 2, 20–29.
- [17] Georg Gottlob and Leonid Libkin, *Investigations on Armstrong Relations, Dependency Inference, and Excluded Functional Dependencies*, Acta Cybernetica **9** (1990), no. 4, 385–402.
- [18] Russel Greiner, Barbara A. Smith, and Ralph W. Wilkerson, *A Correction to the Algorithm in Reiter's Theory of Diagnosis*, Artificial Intelligence **41** (1990), 79–88.
- [19] D. Gunopulos, R. Khardon, H. Mannila, and H. Toivonen, *Data mining, hypergraph transversals, and machine learning*, Proceedings of the 16th ACM SIGACT SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-96), 1993, pp. 209–216.
- [20] Toshihide Ibaraki and Tiko Kameda, *A theory of coterries: Mutual exclusion in distributed systems*, IEEE Trans. Parallel Distrib. Syst. **4** (1993), no. 7, 779–794.
- [21] David S. Johnson, *A Catalog of Complexity Classes*, Handbook of Theoretical Computer Science (J. van Leeuwen, ed.), vol. A, Elsevier Science Publishers B.V. (North-Holland), 1990.
- [22] D. Kavvadias, Ch.H. Papadimitriou, and M. Sideri, *On Horn Envelopes and Hypergraph Transversals*, Proceedings 4th International Symposium on Algorithms and Computation (ISAAC-93) (Hong Kong) (W. Ng, ed.), LNCS 762, Springer, December 1993, pp. 399–405.
- [23] Dimitris J. Kavvadias and Elias C. Stavropoulos, *Monotone Boolean Dualization is in co-NP[log²n]*, Information Processing Letters **85** (2003), 1–6.
- [24] L. Lamport, *The implementation of reliable distributed multiprocess systems*, Comp. Networks **2** (1978), 95–114.
- [25] Christos H. Papadimitriou, *Computational complexity*, Addison-Wesley, 1994.
- [26] Raymond Reiter, *A theory of diagnosis from first principles*, Artif. Intell. **32** (1987), no. 1, 57–95.
- [27] Ken Satoh and Takeaki Uno, *Enumerating maximal frequent sets using irredundant dualization*, Discovery Science (Gunter Grieser, Yuzuru Tanaka, and Akihiro Yamamoto, eds.), Lecture Notes in Computer Science, vol. 2843, Springer, 2003, pp. 256–268.
- [28] H. Tamaki, *Space-efficient enumeration of minimal transversals of a hypergraph*, IPSJ-AL **75** (2000), 29–36.